Bujar Sefa
CSC 21200 - BC
Professor Liu
HW #3

Overview of class:
The node class is a new class that allows users to create, store, and manipulate a node. A node is an object that contains data, in my case integer, and a link field which is a pointer to another node. Nodes connected in a list create what's called a linked list, a list of nodes all with particular data and pointers linking them. The linked list has a head_ptr or head pointer which points to the first node in the list. Nodes then are chained off after. Certain functions can be used to add nodes anywhere in the list, remove nodes in a list, and even swap their data and positions.

/* NOTE I will mention the term data and mean the integer data_field that is stored in each node. I will also say link and mean the link that is stored in link_field. */

1a. Node():
What: This function creates a node.
How: This function works by calling the function via set arguments, being a integer data value, and a link to the next node. Note if arguments are not provided, the the constructor provides a default link of NULL and a default value from the datatypes constructor, in this case 0 for int.
Pre-Condition: N/A
Post-Condition: A node is created.
Worst Time Complexity: O(1)

```
node(const nodeDataType& init_data = nodeDataType(),
   node* init_link = NULL){
   link_field = init_link;
   data_field = init_data;
 }
```

1bi. set_link():
What: This function allows callers to change the link of a node.
How: This function works by editing the member variable link_field and setting it to the nodes new link.
Pre-Condition: N/A
Post-Condition: The node that calls the function is now linked to a new place/node/NULL.
Worst Time Complexity: O(1)

```
void set_link(node* ptr){ link_field = ptr;}
```

1bii. data():
What: This function gets the data of the node.
How: This function works by accessing the nodes member variable data_field and returning the data.
Pre-Condition: N/A
Post-Condition: The data of the node that calls it is returned.
Worst Time Complexity: O(1)

```
nodeDataType data() const{return data_field; }
```

1biii. link():
What: This function gets the link of the node.
How: This function works by accessing the node's member variable link_field and returning the link.
Pre-Condition: N/A
Post-Condition: The link of the node is returned.
Worst Time Complexity: O(1)

```
node* link(){ return link_field; }
```

1biv. const_link():
What: This function gets the link of the constant node.
How: This function works by accessing the const node's member variable link_field and returning the link.
Pre-Condition: The node that calls this function must be constant.
Post-Condition: The link of the const node is returned.
Worst Time Complexity: O(1)

```
const node* link() const{ return link_field; }
```

1bv. set_data():
What: This function allows callers to change the data of a node.
How: This function works by editing the member variable data_field and setting it to the node's new data.
Pre-Condition: N/A
Post-Condition: The node that calls now contains new data.
Worst Time Complexity: O(1)

```
void set_data(const nodeDataType& new_data){ data_field = new_data;}
```

2a. head_insert():

What: This function places a new node into the front of the linked list. Function callers also provide the data that they want for the node. The head_pointer of the linked list points to this new first node.
How: This function works by creating a new node, using the node constructor. The data_field becomes the data that the function caller provides, and the link_field becomes the link of the first head_pointer (which could also be NULL and create the very first node).
Pre-Condition: Entry must be of nodeDataType. HeadPointer must be the head pointer of a linked list.
Post-Condition: The linked list now contains a new head pointer with data that was used from the arguments. If no linked list existed before, now a new one is created with this first new node.
Worst Time Complexity: O(1)

```
void list_head_insert(node*& head_ptr, const node::nodeDataType& entry){
   head_ptr = new node(entry, head_ptr);
}
```

2b. head_remove():
What: This function removes the first node in the linked list (if not NULL).
How: This function works by setting a temporary node as the current head pointer. Then the head pointer is replaced by the second node in the list. Finally the temporary node that contains the original head pointer is deleted.
Pre-Condition: Head_ptr is a linked list with at least 1 node.
Post-Condition: If a list exists, the second node becomes the head_pointer and the original head pointer is deleted. If the list is NULL, nothing happens.
Worst Time Complexity: O(1)

```
void list_head_remove(node*& head_ptr){
   if(head_ptr == NULL)
       return;
   node* remove_ptr;
   remove_ptr = head_ptr; /*Set node to remove to our current head*/
   head_ptr = head_ptr->link(); /*Make head look at the second node*/
   delete remove_ptr; /*remove old head*/
}
```

2c. end_insert():
What: This function adds a new node at the end of the linked list allowing callers to enter the data of a node.
How: This function works creating a new node, using the constructor, and setting its link to NULL, and entry to what was passed in the arguments. The last node in the current list then has its link set to the new node that was created.
Pre-Condition: Entry must be of nodeDataType. Head_ptr is the head pointer to a linked list.

Post-Condition: The linked list now contains a new end pointer with data that was used from the arguments. If no linked list existed before, now a new one is created with this first new node.
Worst Time Complexity: O(n)

```
void list_end_insert(node *& head_ptr, const node::nodeDataType& entry){
        if(head_ptr==NULL){
               list_head_insert(head_ptr, entry);/*Make a new head and save it*/
               return;
        }
        node * cursor = head_ptr;
        for(cursor; cursor->link()!=NULL; cursor = cursor->link()){
        }
        list_insert(cursor, entry);
  }
```

2d. end_remove():
What: This function removes the last node in the linked list.
How: This function first checks if a list is not NULL. If null, do nothing. Then it iterates to the second to last node. It sets the second to lasts node's link to NULL (what the last node was linked to), and it then calls delete on the last node.
Pre-Condition: Head_ptr is the head pointer to the linked list.
Post-Condition: The linked list now has the second to last node as its last node as the last node was removed.
Worst Time Complexity: O(n)

```
void list_end_remove(node*& head_ptr){
   if(head_ptr == NULL){
       return;
   }
   if(head_ptr->link()==NULL){
       head_ptr->set_link(NULL);
       delete head_ptr->link(); /*Remove last*/
       return;
   }
   node * end = head_ptr;
   node * cursor = head_ptr;
   for(cursor; cursor->link()!=NULL; cursor = cursor->link()){
       end = cursor;
   }
   end->set_link(NULL);/*Set previos to NULL*/
   delete cursor; /*Remove last*/
}
```

2e. size():
What: This function computes the size of the linked list.

How: This function works by iterating through each node and checking that it is not NULL, and incrementing variable length each time for the length/size.
Pre-Condition: head_ptr is the head pointer to the linked list.
Post-Condition: The length/size of the linked list is returned.
Worst Time Complexity: O(n)

```
size_t list_length(const node* head_ptr){
   size_t length = 0;
   for(const node* cursor = head_ptr; cursor != NULL; cursor = cursor->link()){
        length++;
   }
   return length;
}
```

2f. deleteAll():
What: This function deletes all nodes in the linked list.
How: This function works removing the head note each time as it traverses through the list.
Pre-Condition: head_ptr is the head pointer to the linked list.
Post-Condition: All the nodes of the list are deleted.
Worst Time Complexity: O(n)

```
void list_clear(node*& head_ptr){
   while(head_ptr!=NULL){
        list_head_remove(head_ptr);
   }
  }
```

2g. print():
What: This function prints the data of all the nodes in the list.
How: This function works by traversing through the list and getting and counting the data of each node.
Pre-Condition: head_ptr is the head pointer to the linked list.
Post-Condition: The data of each note in the linked list has been printed to standard output.
Worst Time Complexity: O(n)

```
void list_print(const node* head_ptr){
   if(head_ptr == NULL)
        return;
   for(const node* cursor = head_ptr; cursor!=NULL; cursor = cursor->link()){
        cout << cursor->data() << endl;
        /*cout << cursor->link() << endl;*/
   }
 }
```

2h. iPos_insert():

What: This function adds a new node at the ith position of the linked list allowing callers to enter the data of a node.

How: This function works by locating the node previous to the ith position, and the creating a new node that links the the node at the ith position and having the node previous to the ith position link to this new node.

Pre-Condition: head_ptr is the head pointer to the linked list. pos is a position in the length of the linked list.

Post-Condition: A linked list now has a new node at the ith position. If ith was out of bounds, the new node is added as the head pointer.

Worst Time Complexity: O(n)

```
void list_ith_insert(node *& head_ptr, const node::nodeDataType& entry, const size_t& pos){
        size_t len = list_length(head_ptr);
        size_t i = pos;
        if(head_ptr == NULL || pos ==0){
               list_head_insert(head_ptr, entry);
               return;
        }
        if(i > len){
               i = len;
        }
        node * add = list_locate(head_ptr, i);/*NOTE i-1 to find previous to use insert*/
        list_insert(add, entry);
    }
```

2i. iPos_delete():

What: This function deletes the node at the ith position.

How: This function works by locating the node previous to the ith position, and linking it to the node following the ith position. Then the node at the ith position is deleted. If position is out of bounds, the head node is removed.

Pre-Condition: head_ptr is the head pointer to the linked list.

Post-Condition: The linked list now is a list without the ith node. If ith node was out of bounds, the head node has been removed.

Worst Time Complexity: O(n)

```
void list_ith_remove(node* head_ptr, const size_t& i){
        size_t pos = i;
        size_t len = list_length(head_ptr);
        if(pos>len){
               pos = len; /*Set to last position.*/
        }
        if(head_ptr == NULL)
               return;
         if(pos == 0){
               list_head_remove(head_ptr);
               return;
        }
        node * beforeIth = list_locate(head_ptr, pos);
               list_remove(beforeIth);

    }
```

2j. overload_insert(): (Code for head, end, ith)

What: This function overloads the head, end, ith insert function with using a second node instead of a constant entry as the second parameter.

How: This function works by first checking if the second node, is NULL. If it is not null, the it calls the head/end/ith insert functions with new_node->data() as the entry.

Pre-Condition: head_ptr is the head node to the linked list. New_node is a node with data used to insert.

Post-Condition: The node now has a new head/end/ith node with its data from new_node.

Worst Time Complexity: Complexity of head_insert, end_insert, or ith_insert ^above.

```
void list_head_insert(node*& head_ptr, node* new_head){
   if(new_head ==NULL)
       return;
   head_ptr = new node(new_head->data(), head_ptr);
}

void list_end_insert(node *& head_ptr, node* new_end){
  if(new_end == NULL)
   return;
  list_end_insert(head_ptr, new_end->data()); /*Checks if head is null*/
}

void list_ith_insert(node *& head_ptr, node* new_ptr, const size_t& pos){
   if(new_ptr == NULL)
       return;
   list_ith_insert(head_ptr, new_ptr->data(), pos);
}
```

2k. (Both const and not const) locate():

What: This function finds a node at a given position.

How: This function works by traversing through each node in the linked list and incrementing position until if find the position from the argument.

Pre-Condition: head_ptr is the head pointer to the linked list. Position must be within the length of the linked list.

Post-Condition: The node at the specified position is returned. If the position is out of bounds, NULL is returned.

Worst Time Complexity: O(n)

```
node* list_locate(node * head_ptr, const size_t& position){
   node * cursor = head_ptr;
   assert(0 < position);
   size_t i = 1; /*NOTE start from 1 because thats where head_ptr starts.*/
   for(i; i< position && cursor!=NULL; i++){
        cursor=cursor->link();
   }
        return cursor;
 }

const node* list_locate(const node* head_ptr, const size_t& position){
   const node * cursor = head_ptr;
   assert(0 < position);
   size_t i = 1; /*NOTE start from 1 because thats where head_ptr starts.*/
   for(i; i< position && cursor!=NULL; i++){
        cursor=cursor->link();
   }
   return cursor;
}
```

2l. (Both const and not const) search():

What: This function finds a node at a given data_field.

How: This function works by traversing through each node in the linked list and checking if the data in the note matches the target data.

Pre-Condition: head_pointer is the head pointer of the linked list.

Post-Condition: Returns the node which contains target data. If target data is not in the linked list, NULL is returned.

Worst Time Complexity: O(n)

```
node * list_search(node* head_ptr, const node::nodeDataType& target){
   if(head_ptr == NULL){
        return NULL;
   }
   for(node* cursor = head_ptr; cursor!=NULL; cursor = cursor->link()){
        if(cursor->data() == target)
             return cursor;
   }
   return NULL; /*If didn't find anything, return NULL*/
}

const node* list_search(const node* head_ptr, const node::nodeDataType& target){
   if(head_ptr == NULL)
        return NULL;
   for(const node* cursor = head_ptr; cursor!=NULL; cursor = cursor->link()){
        if(cursor->data() == target)
           return cursor;
   }
   return NULL;
}
```

2m. cycle():

What: This function checks to see if the nodes in the linked list form a cycle, meaning there is no end to the list.

How: This function works by checking if any of the links in the linked list are repeated. There are two pointers one that starts in first position, second that start two positions after. First traverses by 1, the second traverses by 2. If there is a cycle they will eventually fall on the same node.

Pre-Condition: head_ptr is the head pointer to the linked list.

Post-Condition: The function returns whether or not the linked list forms a cycle.

Worst Time Complexity: O(n)

```
bool list_cycle(const node* head_ptr){
 if(head_ptr == NULL || head_ptr->link() == NULL || head_ptr->link()->link() == NULL){
        return false;
 }
 const node * firstCursor = head_ptr;
 const node * secondCursor = head_ptr->link();
 secondCursor = secondCursor->link();
 while(secondCursor != NULL && firstCursor != NULL){
        if(secondCursor == firstCursor)
                return true;
        secondCursor = secondCursor->link();
        if(secondCursor == NULL)
                return false;
        secondCursor= secondCursor->link();
        firstCursor = firstCursor->link();
 }
 return false;
}
```

2n. swap_i_i+1():

What: This function swaps the node at position i and i+1.

How: This function works by finding the node previous to position i using the locate function. It the swaps the links of the node at i and i+1. The node previous of i links to the one following i, and the one following i links to i, and i links to the one that the following links to.

Pre-Condition: head_ptr is the head pointer to the linked list. I_pos is a position in the linked list length - 1.

Post-Condition: The link list now has the node at i and i+1 swapped.

Worst Time Complexity: O(n)

```
void list_swap_next(node*& head_ptr, const std::size_t& i_pos){
   size_t len = list_length(head_ptr); /*O(n)*/
   if(i_pos+1>len-1 || len < 2)
       return;
   if(head_ptr == NULL || head_ptr->link() == NULL){
       return;
   }
   if(i_pos == 0){
       node * iNext = head_ptr->link();
       head_ptr->set_link(iNext->link());
       list_head_insert(head_ptr, iNext->data());
       delete iNext;
       return;
   }
   node * i_previous = list_locate(head_ptr, i_pos);
   node * i_node = i_previous->link();
   if(i_previous->link() == NULL || i_node->link()==NULL)
       return;
   node * i_next = i_node->link();
   node * temp = i_node;
   i_previous->set_link(i_next);
   i_node->set_link(i_next->link());
   i_next->set_link(temp);
 }
```

2o. swap_i_j():

What: This function swaps the node at position i and j.

How: This function works by finding the node previous to position i, j using the locate function. It the swaps the links of the node at i and j. The node previous of i links to the j, the node j links to the node following i; the node i links to the node following j, and the node previous of j links i.

Pre-Condition: head_ptr is the head pointer to the linked list. I and J are positions in the bounds of the linked list.

Post-Condition: The nodes i and j are swapped in the linked list.

Worst Time Complexity: O(n)

```
void list_swap_IJ(node *& head_ptr, const std::size_t& i_pos, const std::size_t& j_pos){
        size_t len = list_length(head_ptr);
        if(j_pos> len || i_pos > len) /*If trying to swap with something out of bounds*/
               return;
        if(head_ptr == NULL) /*If no list, return*/
               return;
        if(i_pos==j_pos)/*If they are the pointing to the same pointer*/
               return;
        if((j_pos>i_pos) && ((j_pos-i_pos) == 1)){ /*If they are next to each other call swap
next*/
               list_swap_next(head_ptr, i_pos);
               return;
        }
        if((i_pos>j_pos) && ((i_pos-j_pos)==1)){ /*If they are next to each other call
swap_next*/
               list_swap_next(head_ptr, j_pos);
               return;
        }
        /*No need to swap links if only switching head with any position.*/
        if(i_pos ==0){
               node* j_prev = list_locate(head_ptr, j_pos);
               node * j_ptr = j_prev->link();
               node::nodeDataType temp = j_ptr->data();
               j_ptr->set_data(head_ptr->data());
               head_ptr->set_data(temp);
               return;
        }
        if(j_pos ==0){
               node* i_prev = list_locate(head_ptr, i_pos);
               node * i_ptr = i_prev->link();
               node::nodeDataType temp = i_ptr->data();
               i_ptr->set_data(head_ptr->data());
               head_ptr->set_data(temp);
               return;
        }
        node * i_prev = list_locate(head_ptr, i_pos);
        node * i_ptr = i_prev->link();
        node * i_next = i_ptr->link();
        node * j_prev = list_locate(head_ptr, j_pos);
        node * j_ptr = j_prev->link();
        node * j_next = j_ptr->link();
        j_prev->set_link(i_ptr);
        j_ptr->set_link(i_next);
        i_ptr->set_link(j_next);
        i_prev->set_link(j_ptr);

    }
```

2p. reverse():

What: This function reverses the nodes in the linked list

How: This function works by creating a new list with all the nodes reversed and then setting head_ptr to the new list.

Pre-Condition: head_ptr is the head pointer to the linked list.
Post-Condition: The linked list has been reveresed.
Worst Time Complexity: O(n)

```
void list_reverse(node*& head_ptr){
   node* reversed = NULL;
   for(node*cursor = head_ptr; cursor!=NULL; cursor=cursor->link())
       list_head_insert(reversed, cursor->data());
   head_ptr = reversed;
 }
```

Questions or concerns:

1. I think I might have messed up on the swapping functions/insert functions in which I start from position 0 instead of position 1. Head being 0 not 1. I was a bit confused as in locate it says to start from position 1. With that said, if it is supposed to start from position 1, I would include a assert(pos < 0); Then where I have if statements checking if pos == 0 and treating them as head pointers, I would do this for pos == 1. And then all other cases of pos would also be shifted by 1. (NOTE by pos I mean i_pos/j_pos etc).
2. Not sure if for reverse I was supposed to delete old list.
3. I made the decision to remove/insert the head node when i position for ith insert was out of bounds. This can easily be replaced with NULL, just do nothing.

Helper methods not mentioned:

```
void list_insert(node* previous_ptr, const node::nodeDataType& entry){
   node* insert_ptr = new node(entry,previous_ptr->link());
   previous_ptr->set_link(insert_ptr);
}

void list_remove(node* previous_ptr){
   node* remove_ptr;
   remove_ptr = previous_ptr->link(); /*Create a new node that points to current*/
   previous_ptr->set_link(remove_ptr->link()); /*Make previous node point to next*/
   delete remove_ptr; /*delete current*/
}
```

Header File: Sefa_Bujar_HW3.h:

```
#ifndef __NODE_H__
#define __NODE_H__

#include <cstdlib>
#include <iostream>
#include <cassert>
using namespace std;
```

```
class node{
    public:
        typedef int nodeDataType;
        void set_link(node* ptr){ link_field = ptr;}
        nodeDataType data() const{return data_field; }
        node* link(){ return link_field; }
        const node* link() const{ return link_field; } /*NOTE this function is constant
because a constant pointer can call it, where it doesn't want to change any of its values in
the future. */
        node(const nodeDataType& init_data = nodeDataType(),
                       node* init_link = NULL){
               link_field = init_link;
               data_field = init_data;
        }
        void set_data(const nodeDataType& new_data){ data_field = new_data;}
    private:
        node* link_field;
        nodeDataType data_field;
};

    void list_clear(node*& head_ptr); /* NOTE *&   tells us that we want to be able to change
the location of head_ptr, ie what it points to. */
    void list_head_insert(node*& head_ptr, const node::nodeDataType& data);
    void list_head_insert(node*& head_ptr, node * new_head);
    void list_insert(node* previous_ptr, const node::nodeDataType& data);
    void list_end_insert(node *& head_ptr, const node::nodeDataType& entry);
    void list_end_insert(node*& head_ptr, node * new_tail);
    void list_ith_insert(node *& head_ptr, const node::nodeDataType& entry, const size_t&
pos);

    void list_ith_insert(node *& head_ptr, node* new_ptr, const size_t& pos);
    size_t list_length(const node* head_ptr);
    void list_remove(node* previous_ptr);
    void list_head_remove(node*& head_ptr);
    void list_end_remove(node*& head_ptr);
    void list_ith_remove(node* head_ptr, const size_t& i);

    void list_copy(const node* source_ptr, node *& head_ptr, node *& tail_ptr);
    node * list_locate(node* head_ptr, const size_t& position);
    const node* list_locate(const node* head_ptr, const std::size_t& position);
    node * list_search(node* head_ptr, const node::nodeDataType& entry);
    const node * list_search(const node* head_ptr, const node::nodeDataType& entry);
    void list_tail_remove(node* ptr); /*NOT sure which pointer would be entered here.*/
    void list_print(const node* head_ptr);
    bool list_cycle(const node* head_ptr);
    void list_swap_next(node*& head_ptr, const std::size_t& i_pos);
    void list_swap_IJ(node *& head_ptr, const std::size_t& i_pos, const std::size_t& j_pos);
    void list_reverse(node*& head_ptr);

#endif
```